Task-Based Programming with OmpSs and Its Application

Alejandro Fernández[1], Vicenç Beltran[1], Xavier Martorell[1,3], Rosa M. Badia[1,2], Eduard Ayguadé[1,3], and Jesus Labarta[1,3]

[1] Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS), Spain
{alejandro.fernandez,vicenc.beltran,xavier.martorell, rosa.m.badia,eduard.ayguade,jesus.labarta}@bsc.es,
[2] Artificial Intelligence Research Institute (IIIA), Spanish Council for Scientific Research (CSIC), Spain
[3] Universitat Politècnica de Catalunya, Spain

Abstract. OmpSs is a task-based programming model that aims to provide portability and flexibility for sequential codes while the performance is achieved by the dynamic exploitation of the parallelism at task level. OmpSs targets the programming of heterogeneous and multi-core architectures and offers asynchronous parallelism in the execution of the tasks. The main extension of OmpSs, now incorporated in the recent OpenMP 4.0 standard, is the concept of data dependences between tasks.

Tasks in OmpSs are annotated with data directionality clauses that specify the data used by it, and how it will be used (read, write or read&write). This information is used during the execution by the underlying OmpSs runtime to control the synchronization of the different instances of tasks by creating a dependence graph that guarantees the proper order of execution. This mechanism provides a simple way to express the order in which tasks must be executed, without the need of adding explicit synchronization.

Additionally, OmpSs syntax offers the flexibility to express that given tasks can be executed on heterogeneous target architectures (i.e., regular processors, GPUs, or FPGAs). The runtime is able to schedule and run these tasks, taking care of the required data transfers and synchronizations. OmpSs is a promising programming model for future exascale systems, with the potential to exploit unprecedented amounts of parallelism while coping with memory latency, network latency and load imbalance.

The paper covers the basics of OmpSs and some recent new developments to support a family of embedded DSLs (eDSLs) on top of the compiler and runtime, including an prototype implementation of a Partial Differential Equations DSL.

1 Introduction

During the last decades, the number of available transistors inside a chip has continuously increased as predicted by the well known Moore's law [12]. The

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 601–612, 2014.

extra transistors provided by each successive processor generation have been traditionally used to increase the complexity of the processors and the size of the cache memories. However, due to the memory and power walls, this trend has halted and replaced by the multi-core and heterogeneous era.

Multi-core processors and heterogeneous architectures are still quite complex, with several functional units in them, including floating point units and vector units. Also, the ability to place other accelerators in the same chip or connected through the PCI express bus resulted in heterogeneous computing nodes. Examples of these architectures are the Xeon Phi processor or general purpose processors with GPU cards. While this trend has been observed for about a decade now, the difficulty to program such architectures still represents a challenge.

Additionally, the interface to program a processor has increasingly been complicated with specific instructions for vector units, specific languages for accelerators which include calls to APIs for data allocation and management (i.e. CUDA or OpenCL), APIs for offloading computation, etc.

All this specific code requirements have made the life of programmers increasingly more difficult, forcing them to mix application logic with specialized instructions. Such code complexity is inversely proportional to code readability and maintainability, thus resulting in an undesired trade-off between productivity and performance. Moreover, these programs are hardly portable: every time a new architecture appears, a new version of the code is necessary. For example, a large number of applications has recently been adapted to enable their execution in nodes with GPUs.

Fig. 1. Software stack in the BSC vision

In this situation, strategies to offer higher levels of abstraction to application developers are necessary. The specifics of the different architectures and hardware organization (architecture-dependent instructions, APIs, separate memory spaces, etc.) should be hidden from the application developers, enabling them to focus on the logic of the application rather than on the low-level performance aspects.

Figure 1 illustrates this idea where a higher level interface in the form of a programming model is offered to the applications. With this layer, a cleaner, more abstract interface results in clean programs without hardware-specific details. Such abstraction is possible thanks to an underlying compiler and/or runtime infrastructure which is the responsible for dealing with the APIs and specific features of the hardware.

In the case of the Barcelona Supercomputing Center (BSC), the programming model considered is StarSs[1], a task-based programming model with tasks' data dependencies taken into account at execution time, building a task dependence graph which defines a partial execution order of the tasks. While the sequential programming paradigm with information about the tasks and the directionality of its parameters is the user interface[2], applications are executed in parallel thanks to the information about the potential parallelism that is derived from the task graph. Another feature of the StarSs programming model is that it enables the application to be unaware of the underlying computing platform. For example, in StarSs instances tailored for distributed computing, the runtime will be responsible for the corresponding data transfers required between computing nodes, performing these activities in a way transparent to the application.

Additionally, in order to offer an even higher level of abstraction, the construction of a high performance framework for a family of Domain-Specific Languages (DSLs) on top of the programming model is currently being considered. DSLs are a promising approach to hide the complexity of hardware systems and boost programmers' productivity. However, the huge cost and complexity of implementing efficient and scalable DSLs, specially for complex platforms such as HPC systems, is hindering their adoption for most domains. For this reason, the strategy adopted at BSC has been to divide the complexity of building such a programming interface by building a DSL development infrastructure on top of one of the implementations of the StarSs programming model. Each instance of this DSL family can focus on a different domain and can be of a different level of complexity (different sizes of DSLs boxes in Figure 1 represent this heterogeneity).

This paper will review the current status of one of the StarSs implementations, the OmpSs project, as well as present an overview of the recent developments towards DSLs for HPC environments. The rest of the paper is structured as follows:

[1] StarSs stands for Star superscalar, since most of the ideas behind this programming model are inspired by the field of computer architecture and superscalar processors.

[2] By directionality we mean, `input` when the parameter is read, or `output` when the parameter is written. This information is used at runtime to derive the data-dependences between tasks.

First, Section 2 presents the StarSs programming model and its instance, OmpSs. Then, Section 3 presents the OmpSs programming model. Next, Section 4 presents the DSL family developed on top of the OmpSs infrastructure and Section 5 concludes the paper.

2 StarSs Overview

StarSs is a family of programming models recently developed at BSC. The main characteristics of these programming models are: task–based programming with indication of data directionality, flat single logical address space, and a dynamic behaviour addressed by a runtime that takes care of functions such as generation of a task-dependence graph, task scheduling driven by the partial order defined by this graph, resource selection, automatic data transfers, etc.

Several prototype implementations of these programming models have been developed to test main ideas in different computing platforms and to make progress in research topics, the more relevant being: GRIDSs [3] for grid computing, CellSs [14] for the Cell processor, SMPSs [13] for shared memory systems, and GPUSs [2] for heterogeneous nodes with GPUs.

BSC efforts currently focus in two implementations: OmpSs [7], for HPC (multicore and heterogeneous computing), and COMPSs [19] for distributed computing and cloud computing.

This paper focuses in the OmpSs implementation, which merges the OpenMP standard [1] with the StarSs extensions. OmpSs has been used to promote the StarSs ideas (tasking, dependences, support to heterogeneity) into the OpenMP standard. Achievements of the BSC team in this aspect have been the inclusion of the tasking model (version 3.0) and dependences in tasks (version 4.0). However, OmpSs does not intend to be a reference implementation of OpenMP, but a long term research project where new ideas can be evaluated.

Currently OmpSs features which do not have a match in the OpenMP standard include the support of non-contiguous/strided regions in their dependence detection and data-management mechanisms. The OpenMP dependence mechanism uses the initial address of a region to detect dependences between tasks and therefore dependences between partially overlapping regions or strided regions cannot be detected [6].

Support of heterogeneity in OpenMP 4.0 and in OmpSs is significantly different and complementary. While OmpSs extensions to support heterogeneous environments are designed to simplify the synchronization and data transfers required between host and accelerator codes, OpenMP tries to generate parallel kernels from annotated sequential code that can efficiently run on accelerators. While both OpenMP and OmpSs specifications include a `target device` clause, this clause has a different semantics in OpenMP and OmpSs. In OmpSs, the options of clause `target device` are, for example, `cuda` or `opencl`, while in OpenMP the clause takes a numeric parameter, e.g., `target device (3)`, which represents a device. In OmpSs, the programmer needs to provide the code of the kernel in CUDA or OpenCL, but this code can be part of a task, and

therefore it will be independently scheduled and executed asynchronously on a device [7], [9]. The support provided by OmpSs includes the ability to schedule tasks in multiple GPUs independently of the code and automatic data transfers (including awareness of the data locality to reduce the number of transfers). In OpenMP, the compiler translates the C code to the language required by the device (i.e. CUDA), but the code is bound to a given device specified statically in the clause, and the programmer is responsible for finding the identifier of the device. In terms of scheduling, the code embedded in a `target device` clause in OpenMP is executed synchronously, and in case it is embedded in a task which will enable the asynchronous execution, the programmer needs to guarantee the exclusive access to the device at every moment since no support is provided by the OpenMP scheduling.

Another support of scheduling in OmpSs is the possibility of providing more than one implementation (version) of a given task through the `implements` clause. The versions can target one or more devices. At runtime, the scheduler will decide which version should be scheduled taking into account parameters such as execution time or locality of the data. Even more, if slower devices are idling, a few tasks can be scheduled there [15].

The OmpSs runtime is able to target heterogeneous devices not only of a single node, but also of several nodes in a cluster [5]. In this case, the OmpSs scheduler distributes the tasks to the different nodes. As in the case of the GPUs, the required data transfers are performed transparently by the runtime. The runtime keeps a directory with information of the locations of the data regions in the cluster. This directory comes with a software cache policy implemented in each memory space (both memory nodes and GPU memory spaces). Concerning programming methodology, while there are no specific requirements for these architectures, organizing the applications in nested tasks improves the performance. With nested tasks, first level tasks are generated by the main program and scheduled in nodes of the cluster. The node responsible for executing this task will generate the children tasks which are naturally scheduled on the node, including both CPU and GPU tasks.

With regard to the hybrid version of OmpSs with MPI, the strategy goes beyond the traditional parallelization at the node level with OmpSs using MPI for the communication between nodes: with MPI/OmpSs, MPI communications are wrapped into OmpSs tasks which are then automatically included in the task dependence graph. With this approach, overlapping of communication and computation is naturally achieved, since computations that do not hold any dependence with the communication tasks may be executed earlier or together with the communication tasks. Additionally, this implementation presents better sensibility to OS noise and jitter [20].

To further improve the behaviour of MPI/OmpSs applications, DLB is a dynamic library designed to speed up hybrid applications with nested parallelism by improving the load balance each computational node [10]. In general, DLB will redistribute the computational resources of the second level of parallelism (OmpSs) to improve the load balance of the outer level of parallelism (MPI).

This is achieved by dynamically and automatically lending threads between MPI processes sharing the same node.

3 OmpSs Development Environment

OmpSs infrastructure is composed of two main components: Mercurium, the compiler, and Nanos++, the runtime. Mercurium is a source to source compiler that supports C99, C++ 2003 and Fortran 95 and also (an increasing) set of features of C 2011, C++ 2011 and Fortran 2003/2008 and extensions of GNU C/C++/Fortran (see Figure 2). The goal of Mercurium is to provide a sufficiently powerful framework for high-level transformations and analyses in source code in order to support research in parallel and high performance programming models.

In order to support heterogeneous computing, Mercurium supports multi-file processing, that is, from a single source file Mercurium can generate several source files which can be combined at the link step. Compiler phases can reintroduce new files into the compilation pipeline and new files may use a different compilation pipeline.

Mercurium processes the OmpSs pragmas and inserts the corresponding calls to the Nanos++ interface. Mercurium also parses CUDA and OpenCL and emits this code unchanged.

After the compiler phase, the corresponding back-end compiler is invoked. This can be configured to use different compilers (i.e., gcc or icc for C code). For the case of CUDA, the NVIDIA compiler is later invoked. For OpenCL, the

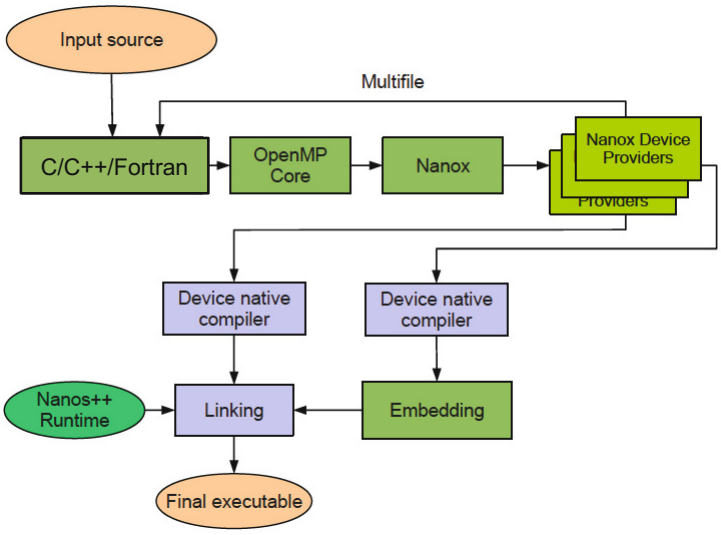

Fig. 2. Mercurium compiler structure

code is processed at execution time by the selected OpenCL runtime. Finally, all objects are linked and embedded into a single binary.

Nanos++ is the OmpSs runtime (see Figure 3). This piece of software is organized in components, each of them responsible for a given behaviour: thread management, task management, dependence checking, cache management, etc. Several of these components are configurable, such as the scheduling policy, the throttle policy or the dependence checker. The runtime also has specific components for the supported devices: SMP, GPU, Cluster, Tasksim (an architecture simulator [16]), etc.

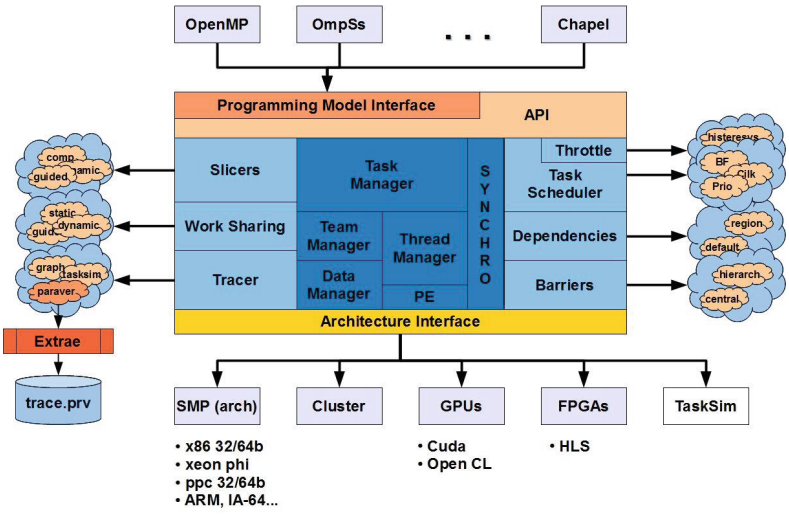

Fig. 3. Nanos++ runtime structure

The runtime can be compiled in different flavours: performance, debug, and instrumentation. While the performance flavour would be the default version to use, the debug version can be used for debugging purposes. The instrumented version is used for several purposes: trace file generation, task graph generation, and debug with Temanejo [18].

The trace file generation emits a time stamped event list ordered by time with information about what happened at execution time. The format of this trace file conforms to the Paraver format (in fact, the Extrae instrumentation library, provided to generate Paraver trace files is called by Nanos++) [11]. Paraver is a very powerful performance visualization and analysis tool based on traces that can be used to analyse any information that is expressed on its input trace format. Its analysis power is based on two main pillars. First, its trace format has no semantics; extending the tool to support new performance data or new programming models requires no changes to the visualizer, just to capture such data in a Paraver trace. The second pillar is that the metrics are

not hard-wired in the tool but programmed. To compute them, the tool offers a large set of time functions, a filter module, and a mechanism to combine two time lines. This approach allows displaying a huge number of metrics with the available data. To be able to analyse OmpSs programs, a set of configuration files is provided with the OmpSs distribution that enable to visualize meaningful views (i.e., view of tasks executed in each thread, communications between host and GPU when running on a GPU node, etc), while each programmer/developer can build up her own configuration files with specific purposes.

Another alternative when running with the instrumentation library is to generate an image of the task dependency graph, which can be later visualized with a PDF viewer. This option is very useful for a quick check by the application programmer about the actual task graph generated.

Both these views will only work if the application is not faulty. In case of a faulty application, the environment provided by the Ayudame and Temanejo libraries can be used [4]. Ayudame is a library which is used to receive information (events) from the Nanos++ runtime system and to exert control over it by issuing requests to it. Temanejo is the graphical front end. It enables to display the task dependency graph of OmpSs applications, and to allow simple interaction with the Nanos++ runtime system in order to control some aspects of the parallel execution of a given application. For example, it enables to execute tasks one at a time or group of tasks, define breakpoints, connect to the GNU debugger to perform a more detailed debug, etc.

4 DSLs on Top of OmpSs

Domain Specific Languages (DSLs) boost programmer productivity by offering experts high level abstractions focused on their domain. With this type of languages, mapping and solving a domain problem becomes extremely easy. Additionally, due to the clarity of the code, applications are easily maintained and extended.

However, developing a DSL is expensive and complex, and therefore it would be only justified when a large community is behind. With this idea in mind, the strategy of the BSC Computer Science department has been to develop a framework that can be shared by several DSLs.

This framework is composed of a HPC compiler framework and a runtime system. The compiler framework is based on Lightweight Modular Staging (LMS) [17] (see Figure 4), a Scala library for embedding DSL compilers together with DSL applications, thus reusing the Scala features to define new languages. LMS is a technique for embedding DSLs as libraries into Scala as a host language, while enabling domain specific optimizations and code generation.

As an intermediate language between the actual DSL and the OmpSs compiler, the Data Flow Language (DFL) [8] has been defined. DFL provides a data-flow model based on four concepts: buffers, tasks, kernels and high-level operations. Buffers abstract the concept of data, while tasks and kernels represent computations written in C++ and OpenCL on a multi-core or accelerator,

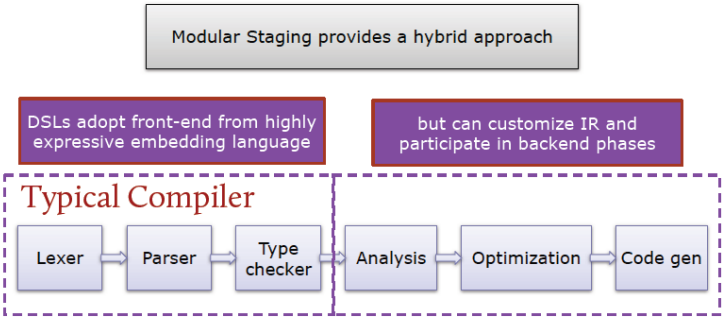

Fig. 4. LMS library design idea

respectively. With these features DFL provides a powerful abstraction to implement HPC DSLs that run on machines composed of CPUs and accelerators.

With the collaboration of the BSC CASE department, Saiph, a DSL for solving Convection-Diffusion-Reaction (CDR) equations has been defined. In this DSL, the programmer first specifies a physical geometry and a set of boundary conditions on that geometry. Then, the initial state of the system is specified by means of functions. Afterwards, the equation to simulate is specified, and the DSL generates DFL and OpenCL code to automatically run the simulation on a multi-GPU architecture.

In addition, some data post process can be specified in order to visualize the output or convert it to a scientific format for analysis tools. An example application of the DSL for CDR equations is shown in Listing 5.

```
1  // Defining preprocess
2  val pre = PreProcess(waveSource1, waveSource2, waveSource3)
3
4  // Defining equation
5  val wavePropagation = c*c * lapla(pressure) - dt2(pressure)
6
7  // Defining postprocess
8  val post = PostProcess(snapshoot each 10 steps)(VTK)
9
10 solve(pre)(post) equation wavePropagation to "wave"
```

Fig. 5. Sample DSL code for a CDR equation

From this input code, the environment generates (see Figure 6) a set of OpenCL kernels that solve the equations and a DFL application that calls the kernels. The DFL application is finally translated to an OmpSs application.

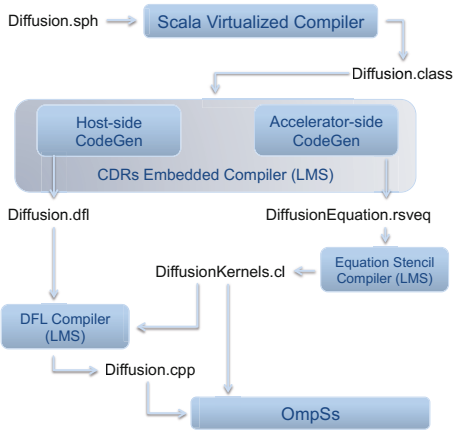

Fig. 6. DSL framework structure

A prototype implementation of this entire framework has been implemented at BSC.

5 Conclusions

This paper has reviewed the current state of the OmpSs programming model, including new developments in the design and implementation of a family of DSLs. While OmpSs offers a reasonable programming interface to average to advanced programmers, more specialized languages will increase the productivity of computational scientists in general. The goal is to achieve high programming productivity with efficient execution in an HPC system.

Acknowledgements. This work has been developed with the support of the grant SEV-2011-00067 of the Severo Ochoa Program, awarded by the Spanish Government, by the Spanish Ministry of Science and Innovation (contracts TIN2012–34557, and CAC2007–00052) and by the Generalitat de Catalunya (contract 2014–SGR-1051).

References

1. OpenMP architecture review board, OpenMP 4.0 specification, `http://www.openmp.org`
2. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An extension of the starSs programming model for platforms with multiple gPUs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009)
3. Badia, R.M., Labarta, J., Sirvent, R., Pérez, J.M., Cela, J.M., Grima, R.: Programming grid applications with grid superscalar. Journal of Grid Computing 1(2), 151–170 (2003)

4. Brinkmann, S., Niethammer, C., Gracia, J., Keller, R.: TEMANEJO - a debugger for task based parallel programming models. In: Proceedings of the ParCO2011 Conference, pp. 639–645 (2011)
5. Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R.M., Ayguade, E., Labarta, J.: Productive Cluster Programming with OmpSs. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 555–566. Springer, Heidelberg (2011)
6. Bueno, J., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Implementing ompss support for regions of data in architectures with multiple address spaces. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 359–368. ACM, New York (2013)
7. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(02), 173–193 (2011)
8. Fernández, A., Beltran, V., Mateo, S., Patejko, T., Ayguadé, E.: A Data Flow Language to Develop High Performance Computing DSLs. In: Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, SC 2014, IEEE Computer Society, New Orleans (2014)
9. Ferrer, R., Planas, J., Bellens, P., Duran, A., Gonzalez, M., Martorell, X., Badia, R., Ayguade, E., Labarta, J.: Optimizing the exploitation of multicore processors and gpus with openmp and opencl. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 215–229. Springer, Heidelberg (2011)
10. Garcia, M., Labarta, J., Corbalán, J.: Hints to improve automatic load balancing with lewi for hybrid applications. J. Parallel Distrib. Comput. 74(9), 2781–2794 (2014)
11. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP: A parallel program development environment. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 665–674. Springer, Heidelberg (1996)
12. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8) (April 1965)
13. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. IEEE Int. Conference on Cluster Computing, 142–151 (September 2008)
14. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM Journal of Research and Development 51(5), 593–604 (2007)
15. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Self-adaptive ompss tasks in heterogeneous environments. In: 27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, pp. 138–149 (2013)
16. Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramírez, A., Valero, M.: Trace-driven simulation of multithreaded applications. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, Austin, TX, USA, April 10-12, pp. 87–96 (2011)
17. Rompf, T., Odersky, M.: Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, pp. 127–136. ACM, New York (2010)

18. Subotic, V., Brinkmann, S., Marjanovic, V., Badia, R.M., Gracia, J., Niethammer, C., Ayguadé, E., Labarta, J., Valero, M.: Programmability and portability for exascale: Top down programming methodology and tools with starss. J. Comput. Science 4(6), 450–456 (2013)
19. Tejedor, E., Badia, R.M.: Comp superscalar: Bringing grid superscalar and gcm together. In: 8th IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2008, pp. 185–193. IEEE (2008)
20. Ayguadé, V.M.J.L.E., Valero, M.: Effective communication and computation overlap with hybrid mpi/smpss. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010. ACM, New York (2010)